



Shanghai Jiao Tong University
Software Theory and Practice Group

Automatic Repair Communication Deadlock

Jinbo Du, Ziyi Lin
2015/11/05

Deadlock

Deadlock describes a situation where **two or more threads** are blocked forever, waiting for each other.


[1]

[1]Java Liveness: <https://docs.oracle.com/javase/tutorial/essential/concurrency/liveness.html>

Deadlock

Deadlock describes a situation where **two or more threads** are blocked forever, waiting for each other.

[1]

Thread(1): {		Thread(2): {
lock(o1);		lock(o2);
lock(o2);		lock(o1);
//		// ...
}		}

[1]Java Liveness: <https://docs.oracle.com/javase/tutorial/essential/concurrency/liveness.html>

Communication Deadlock[1]

```
Thread(1): {      Thread(2): {  
    wait(o);      .. // Exception thrown  
}                notify(o);  
                  }
```

definition: ?

wait: Causes the current thread to wait until another thread invokes the [notify\(\)](#) method or the [notifyAll\(\)](#) method for this object.

notify: Wakes up a single thread that is waiting on this object's monitor.

[1]. Ziyi Lin, JaConTeBe, ASE'15

Java Monitor[1]

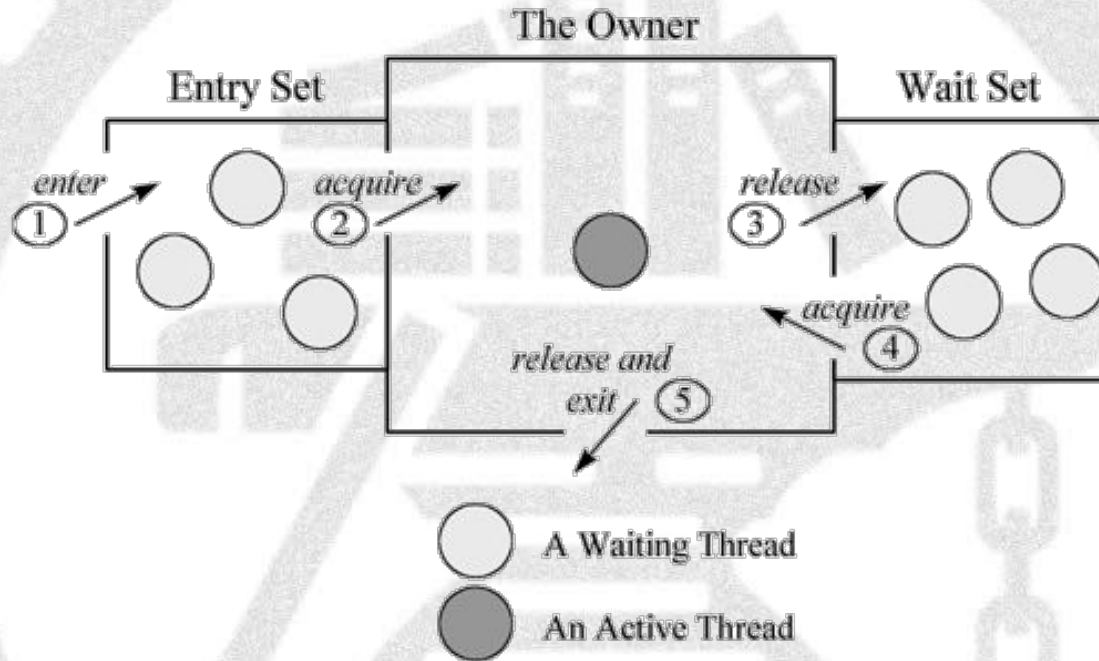


Figure 20-1. A Java monitor.

[1] Inside the Java Virtual Machine, <https://www.artima.com/insidejvm/ed2/threadsynch.html>

Symptoms & Root Causes

- 1. notify-before-wait
 - 2. always-wait
 - 3. notify-not-executed
- A. Scheduling problem
 - B. Redundant locks
 - C. Misuse notify/notifyAll
 - D. Unhandled exceptions
 - E. wrong lib usage
 - F. Logical incorrectness

Symptoms & Root Causes

- 1. notify-before-wait(A)
 - 2. notify-not-executed
 - 3. always-wait
- A. Scheduling problem
 - B. Redundant locks
 - C. Misuse notify/notifyAll
 - D. Unhandled exceptions
 - E. wrong lib usage
 - F. Logical incorrectness

Symptoms & Root Causes

- 1. notify-before-wait(A) A. Scheduling problem
- 2. notify-not-executed(B-F) B. Redundant locks
- 3. always-wait C. Misuse notify/notifyAll
D. Unhandled exceptions
E. wrong lib usage
F. Logical incorrectness

Symptoms & Root Causes

- 1. notify-before-wait(A) A. Scheduling problem
- 2. notify-not-executed(B-F) B. Redundant locks
- 3. **always-wait(F)** C. Misuse notify/notifyAll
D. Unhandled exceptions
E. wrong lib usage
F. Logical incorrectness

Proposed Approach

- Pattern => Repair

* Wrong lib usage not included.

Proposed Approach

- Pattern => Repair
 - notify-not-executed
 - P1: unhandled exception
 - P2: logical problem (miss notify branch)
 - P3: misuse notify/notifyAll
 - P4: redundant locks
 - notify-before-wait
 - P5: scheduling problem

* Wrong lib usage not included.

Pattern 1: unhandled exception

```
Thread(1): {  
    wait();  
}
```

```
Thread(2): {  
    // exception  
    notify();  
}
```

```
Thread(2): {  
    try {  
        // exception  
    } finally {  
        notify();  
    }  
}
```



Pattern 2: missing notify branch

```
Thread(1): {  
    wait();  
}
```

```
Thread(2): {  
    if(condition) {  
        notify();  
    } else {  
        // missing notify  
    }  
}
```



```
Thread(2): {  
    if(condition) {  
        notify();  
    } else {  
        // insert notify  
        notify();  
    }  
}
```

Pattern 3: misuse notify/notifyAll

```
Thread(1): {  
    wait();  
}
```

```
Thread(2): {  
    wait();  
}
```

```
Thread(3): {  
    notify();  
}
```

```
Thread(1): {  
    wait();  
}
```

```
Thread(2): {  
    wait();  
}
```

```
Thread(3): {  
    notifyAll();  
}
```

Pattern 4: redundant locks

<https://bugs.openjdk.java.net/browse/JDK-8012019>



Pattern 5: wait/notify misorder

```
Thread(1): {  
    startThread(2);  
    wait();  
}
```

```
Thread(2): {  
    // do something  
    notify();  
}
```

```
Thread(1): {  
    satisfied = false;  
    startThread(2);  
    while(!satisfied) { [1]  
        wait();  
    }  
}
```

```
Thread(2): {  
    // do something  
    satisfied = true;  
    notify();  
}
```

- [1] waits should always occur in loops, <http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/Object.html>

Threats

- Pattern in small sized samples

Challenge 1

- Communication deadlock identification
 - what exactly are communication deadlocks?

Challenge 1

- Communication deadlock identification
 - what exactly are communication deadlocks?
 - never notify?
 - consider event-based systems: no event?
 - cannot notify?

Challenge 2

- Pattern identification
 - wait-notify pair identification
 - fault localization via wait-notify pairs

Challenge 2

- Pattern identification
 - wait-notify pair identification
 - dynamic: record & replay
 - static: model checking (def-use pair)
 - fault localization via wait-notify pairs

Challenge 2

- Pattern identification
 - wait-notify pair identification
 - fault localization via wait-notify pairs
 - missing notify?
 - notify before wait?
 - notify before exception?
 - (N waits, 1 notify) => notifyAll?
 - etc.

Challenge 3

- Repair real world deadlocks
 - real world fix usually involves a changeset with multiple source locations & refactors

Related Works 1

- Deadlock detect/repair

- Yiyang Lin, Sandeep S. Kulkarni: Automatic repair for multi-threaded programs with Deadlock/Livelock using maximum satisfiability. ISSTA'14
 - repair via CONSTRAINT SOLVING
 - cited 4
- Horatiu Julia, etc. Deadlock immunity: enabling systems to defend against deadlocks. OSDI'08
 - cited 128

Related Works 2

- Mutex pair mismatch
 - Detecting mutex pairs in state spaces by sampling, Mehdi Sadeqi, 2013, cited 2

Related Works 3

- A Study and Toolkit for Asynchronous Programming in C#. Okur, etc. ICSE'14
 - cited by 10
 - C# 5 await/async keywords usage
 - wait/notify vs. await/async

Question?

Thanks

